Advanced model checking for verification and safety assessment
Lab 1

May 22-27, 2016

1 Introduction

This laboratory divided into:

1. Getting started
2. Wolf, Cabbage and Goat Puzzle
3. Redundant Sensor (part 1)

2 Getting started

In this warm-up exercise, you will get familiar with nuXmv and try out some commands.

Every SMV model must have a main module, in which variables are defined. `VAR` define state variables, i.e., variables that have both a current and next value. `IVAR` define input variables (also called transition variables), i.e., variables that are evaluated in the transition between two states; `IVAR`'s are used to encode actions and events.

The following snippet of code defines a simple toggle flip-flop:

```
MODULE main
  VAR
    x: boolean;
  IVAR
    i: boolean;

  INIT !x;
  TRANS i -> next(x) = !x;
  TRANS !i -> next(x) = x;
```

Initially, `x` is false, and every time that the input `i` is set to true, `x` changes value. `INIT` and `TRANS` are used to define the relation between variables in a relational way. This is a powerful way of defining constraints in a declarative way. However, it might be more intuitive to define the evolution of the system in a more imperative way using the `ASSIGN` statement:

```
MODULE main
  VAR
    x: boolean;
  IVAR
    i: boolean;

  ASSIGN
    init(x) := FALSE;
    next(x) := case i : !x;
                    !i: x;
               esac;
```

We can specify invariant properties using `INVARSPEC`, or LTL using `LTLSPEC`. In the above model, add:

```
 LTLSPEC G F(x);
```

Create a file `toggle.smv` with the model above, and the LTL property. Create a file `script.cmd` with the following commands:

```
go_bmc    # Read the model and prepare for SAT based model-checking
check_ltlspec_klive # Verify the LTL property using IC3-Klive
reset     # Reset the state of the system
go_bmc
check_ltlspec_bmc # Verify the LTL property using BMC
quit
```

Scripts can be executed in all the tools (nuXmv, OCRA, xSAP) by running:

```
$ ./nuXmv -source script.cmd toggle.smv
```

Writing scripts is the recommended way of proceeding, since it makes it easier to reproduce the results, and make sure that all steps have been performed. However, sometimes it is useful to experiment with commands. To do so, you can run all the tools in interactive mode:

```
$ ./nuXmv -int toggle.smv
```

Refer to the nuXmv manual (Section 2) for additional information on the modeling language. In particular, you might need to read about enumeration types, `DEFINES`, and `INVAR` for this example. Commands and their options are explained in Section 5.

3 Wolf, Cabbage, and Goat

The power of a model-checker lays in its ability to exhaustively analyze each possible execution of a system. In this exercise, you will model a puzzle using SMV, and then use the model-checking algorithms to find a solution. The puzzle is a well-known puzzle, called "Wolf, Cabbage, and Goat":

> A man has to take a wolf, a goat, and a cabbage across a river. His rowboat has enough room for the man plus either the wolf or the goat or the cabbage. If he takes the cabbage with him, the wolf will eat the goat. If he takes the wolf, the goat will eat the cabbage. Only when the man is present are the goat and the cabbage safe from their enemies. All the same, the man carries wolf, goat, and cabbage across the river. How?

To help you model this problem try to answer the following questions:

- What are the key-concepts of this problem?
- How would you describe an intermediate state of the system?
- How would you describe a solution?

Nouns and state are usually captured using `VAR`s, while actions are usually captured using `IVAR`s.

Once the state-space has been defined, you need to express how the state changes. To simplify debugging of your model, you can either write specifications that you expect to hold, and model-check them. Since this is a simple example, you can also simulate the model:

```
$ ./nuXmv -int wcg.smv
> go # Use BDD-Based engine to simplify simulation
> pick_state # Pick an initial state for the simulation
> show_trace # Show the current
> simulate -i # Start interactive simulation
```

You will be presented with all possible transitions, and you need to specify which one you want to take. Notice that (apart from the first state) all states are expressed as difference with respect to the first state. This visualization is often used to keep the output small.

To simplify the writing of the property, use a `DEFINE` to define the goal state and a target configuration. You want to write a property that is violated by a solution.

4 Redundant Sensor

In this exercise, you will use OCRA to model a Redundant Sensor system. The modeling will be done at the architectural level, and then you will implement the single components using SMV. Using OCRA, you will generate a new SMV model obtained from

the composition of the single components following the architecture, and perform some analysis on it.

4.1 Architecture Overview

The architecture is modeled in a compositional way: each component is defined in terms of its interface with input and output ports specification. The connections within components are defined linking the ports of the components pairing an input port with an output one. A state machine can be associated to each leaf component (i.e. a component not further refined) as behavioral model. The behavior of a composite component (including the top-level component) is given by the composition of such state machines.

In this case the system component (i.e. the top level one) contains three subcomponents: two sensors, and a voter. The voter, in turn, is composed of three monitors and a selector.

All components ports have a domain or type of data associated. The only types used in the model are boolean ($\{TRUE, FALSE\}$) and bounded integers, which represent the readings from the environment and the sensors. The system is at discrete time. All components have an instantaneous reaction of propagation whereas the Selector has a reaction that takes one tick of the system. This is necessary to have the last value to perform the variance check in the monitors. The overall reaction of the system is thus one tick: the corresponding output for a given input is the one generated in the next state.

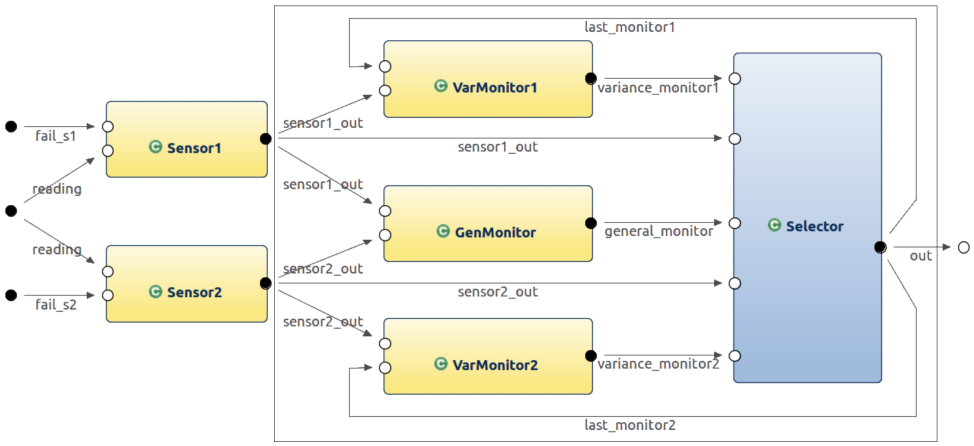

Figure 1: Architecture layout

Component	INPUT	Domain	OUTPUT	Domain
RedundantSensors	reading	value_domain	out	value_domain
Sensors	In	value_domain	out	value_domain
Voter	sensor1	value_domain	out	value_domain
	sensor2	value_domain		
VarMonitor	In	value_domain	Valid	boolean
	Last	value_domain		
GenMonitor	In1	value_domain	Valid	boolean
	In2	value_domain		
Selector	sensor1	value_domain	out	value_domain
	sensor2	value_domain		
	variance_monitor1	boolean		
	variance_monitor2	boolean		
	general_monitor	boolean		

Parameters When developing complex systems, it is possible to use the pre-processor to make the models more understandable and easy to change. It is generally a good idea to abstract the specific data-types, when possible. The domain of the data variables is defined by an integer interval between two fixed bounds ($\{lower\_bound..upper\_bound\}$). The system is then at finite precision values, i.e. the domain of the variables involved in the system is finite. It is important to notice that in the model the parameters are used to specify properties and contracts, in particular: `max_sensor_error`, `max_variance`, `lower_bound` and `upper_bound`. These parameters are contained in the `parameter.h` header file. Note that these parameters are arbitrary chosen but fixed. The system is scalable on all these parameters. In `macros.h`, some additional functions are defined to simplify specification.

Task The file `System.oss` contains part of the architecture defined above. You need to complete it by defining suitable components, and connecting the input/output ports.

4.2 Modeling Overview

We are going to provide an implementation to each leaf component (i.e., a component without refinement).

Sensor The nominal behavior is that at each value in input the sensor adds a non-deterministic but bounded value that simulates the intrinsic error/noise of the sensor. This error is bounded in absolute value at `max_sensor_error`. The sensor adds this error cutting the resulting sum at the domain range and thus the sensor output is always inside the data domain.

General Monitor The general monitor has two input ports that receive the output of the sensors. By comparing the two sensor outcomes, the general monitor implements a failure detection feature in the system. The check is based on the maximum sensor error

assumption: i.e. if the sensors are both correctly working then their reciprocal output difference in the worst case is twice the maximum sensor error, in particular when one sensor error is at `-max_sensor_error` and the other has its error at the other extreme of the error interval `+max_sensor_error`. Using this observation we can say that if the difference of the two sensor outputs is more, in absolute value, than twice the maximum sensor error, then a failure is surely present in one of the two sensors. The output of this monitor is a boolean `Valid` that is equal to true if no failure is detected and false otherwise:

$$Valid := |In1 - In2| \leq 2 * max\_sensor\_error$$

The condition for a failure is only sufficient, so when a failure is detected it is surely present in one of the sensors, but when a failure is not detected not necessarily there is absence of failures.

Variance monitor There is a variance monitor for each sensor. These monitors implement the feature of failure isolation in the system, therefore when a failure is detected by the general monitor then the system tries to isolate which sensor is causing the failure (it is assumed at top level that the sensors are not both failed at the same time). This monitor is based on the assumption regarding the input variance. The variance monitor component is thought to compare the current outcome of a sensor with the last output of the system. Since the system takes one tick to evaluate an input the value comes out in the next step. The variance monitor checks the consistency of a sensor output on the base of its the variance in each tick considering that the system variance is assumed bounded and that the maximum sensor error is fixed. The reasoning is that if the last value is considered reliable then:

$$|real\_last - last| \leq max\_sensor\_error$$

Maximum system error reasoning In the worst case the error in module is equal to the maximum sensor error. If then the input changes as much as possible in a state, i.e. it changes of the value of the maximum variance and in the current state the sensor introduces another worst error, then the overall error is finally given in terms of:

$$error = |last - current| \leq max\_sys\_error$$

where `max_sys_error` is the maximum between:

$$Max\{2 * max\_sensor\_error, max\_sensor\_error + max\_variance\}$$

This is due to the general monitor guarantee, because the last value given as output in a failure case with no isolation has the maximum possible error equal to twice the sensor error. If the maximum variance is less than the maximum sensor error:

$$max\_sensor\_error < max\_variance$$

then the worst case the error is not given by the sum: *max_sensor_error + max_variance*, but by *2*max_sensor_error* since:

$$max\_sensor\_error + max\_variance < 2 * max\_sensor\_error$$

4.3 Selector

This component receives as input the two sensors outputs and the three monitor outcomes. Using this data the selector gives as output:

- the average of the sensors, or
- the value of one sensor, or
- last value given in output;

The average is necessary to maintain the performances because in the worst case the general monitor does not detects a failure and one sensor is far from the real input by its maximum error, then the other sensor is failed but far, in absolute value from the first sensor by at most twice the max sensor error. In this case the overall error is three times the max sensor error, doing the average of the two sensor is reducing the error at twice the maximum sensor error. This component as the name suggests selects the value to be given as output basing on the information from the monitors, that are the failure detection and isolation components. After the hierarchy of the FDI system, the selector primarily controls the general monitor $Valid$ flag, if true then no failure is detected and the output is the average of the two sensors, if false it checks the variance monitor flags trying to determine which sensor is failed and isolate the problem. If one variance monitor valid flag is at false the failure had been isolated on the relative sensor and the output in this case is the non failed sensor value. If none of the two variance monitors isolate the failure then the output is the last value given as output. This last value, when a failure is detected, is reliable because of the assumptions: there are no possible double failures so if a failure is happening in the current state then in the previous state there were no failures at all. Indeed the first state is assumed correct and this reasoning can propagate by induction. Considering that the maximum error of the system in the previous step is (see 4.2):

$$|last - current| \leq max\_sys\_error$$

Then the worst error of the system is bounded at this constant `max_sys_error` that for definition is equal to the maximum between `max_variance + max_sensor_error` and `2*max_sensor_error`.

4.4 Tasks

- Implement the SMV behavior for each leaf component. A stub is given for each component: `Sensor.smv`, `GenMonitor`, `VarMonitor`, `Selector`.
- Take a look at the script `ocra.cmd`. This script performs several checks on the architecture. Moreover, it combines the SMV implementations using the architecture and the nominal component map (`nominal.map`). Run the script using the OCRA tool.

- Under the assumption of **bounded variance** of the input (i.e., sensors inputs do not change too much in one step) what properties are satisfied by the System? Complete the file `nuxmv.cmd` in order to include a specification for the system. file

Bonus Task The types of the ports are defined within the file `parameters.h`. The rest of the model performs arithmetic operations on these values. Can you modify this model to use infinite domain variables?

Hints:

1. The type `real` defines real valued variables;
2. Both the architecture and implementations need to be aware of this change;
3. To verify the model, you need to use infinite state commands, (e.g., `go_msat`).